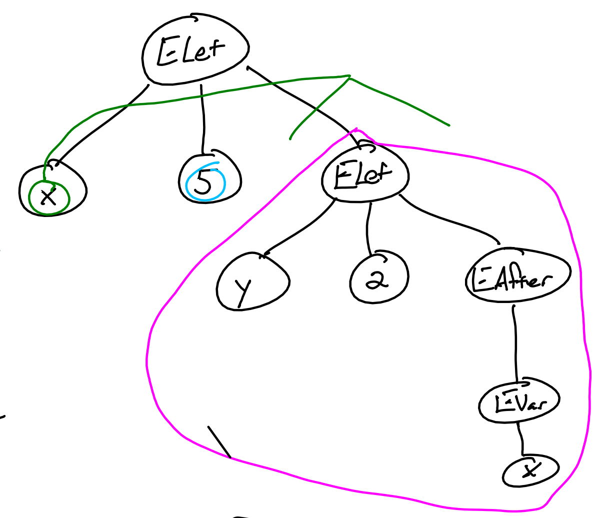
```
type expr =
  | EInt of int
  | EBefore of expr
  | EAfter of expr
  | ELet of string * expr * expr
  | EVar of string
  | ....
```

```
let x = 5 in
let y = 2 in
after(x)
```



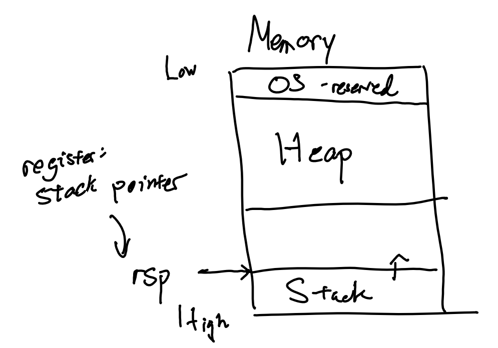
does the same thing that running e would do, leaving the result in RAX

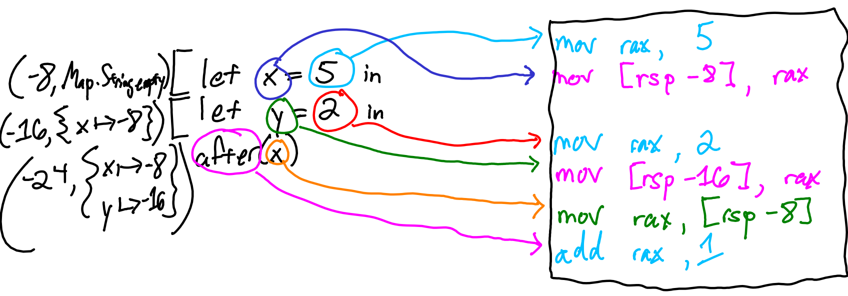
```
let rec compile_expression (e : expr) : instruction list =
  match e with
  | EInt n -> [(mov rax, string_of_int n)]
  | EAfter (e') -> compile_expression e' @ [(add rax, 1)]
```

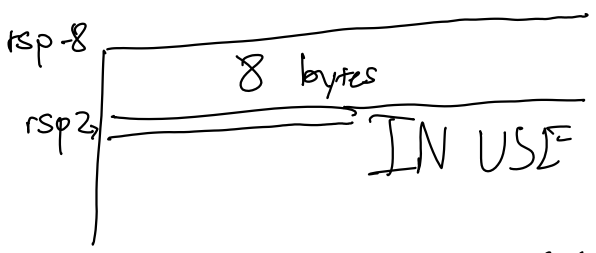
```
mov rax, 2
AsmMov(ArgRegister RAX,
       ArgConst "2")
```

4   mov rax, 4

after(4)   mov rax, 4
                add rax, 1








```
let compile_expression (env : environment) (e : expr) : instruction list =
  match e with
  | ....
  | ELet(x, e1, e2) ->
      ⋮
      let env' = ... in
      let exp2instrs = compile_expression env' e2 in
      ⋮
```

C++
Dictionary<string, argument>

OCaml
int * argument Map.String.t